THE ACTION! TOOLKIT

Helps You Write ACTION! Programs FAST!

Save your valuable programming time! THE ACTION! TOOLKIT! saves you time by giving you a ready-to-use library of ACTION! source code for:

Advanced I/O Functions .. Player/Missile Graphics
Turtle Graphics ..... Floating Point Numbers
Sort Routines ..... Memory Allocation

AND MORE

THE ACTION! TOOLKIT shows you how to write functional ACTION! programs, by giving you full source code to a music maker, complete arcade games, and several demonstration programs.

Requires an Atari Computer with 40KB Memory, Disk Drive, and an ACTION! SuperCartridge.

OSS PRECISION SOFTWARE TOOLS FOR ATARI HOME COMPUTERS

BASIC XL	The most powerful Basic
THE BASIC XL TOOLKIT	Programming Aids
ACTION!	Fastest structured language
THE ACTION! TOOLKIT	Programming Aids
MAC/65	Fastest macro-assembler
THE MAC/65 TOOLKIT	Programming Aids
C/65	A small C language compiler
DOS XL	Now with BUG/65
THE WRITER'S TOOL	Writing was never so natural

Optimized Systems Software, Inc.

1221B Kentwood Avenue, San Jose, California 95129 (408) 446-3099

© 1984 Optimized Systems Software, Inc.

THE Action! TOOLKIT™

An Essential Aid for ALL ACTION! Programmers.

OSS™ Precision Software Tools

A Reference Manual For

The ACTION! ToolKit Diskette

The programs and manual comprising the ACTION! ToolKit
are Copyright (c) 1984 by

Optimized Systems Software, Inc.
Precision Software Tools
1221B Kentwood Avenue
San Jose, CA 95129
(408)446-3099

All rights reserved. Reproduction or translation of any part of this work beyond that permitted by sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Table of Contents

Introduction	1
ABS.ACT	2
ALLOCATE.ACT	3
CHARTEST.ACT	5
CIRCLE.ACT	7
CONSOLE.ACT	8
IO.ACT	9
JOYSTIX.ACT	12
PNG.ACT	13
PRINTF.ACT	17
REAL.ACT	19
SORT.ACT	26
TURTLE.ACT	28
GEM.DEM	30
KALSCOPE.DEM	31
MUSIC.DEM	31
SNAILS.DEM	32
WARP.DEM	33

Introduction

Welcome to the Programmers' Tool Kit. This diskette contains routines written in ACTION! which extend your ACTION! programming capabilities. The following is a list of the files on the disk, together with a short description of what each file does.

ABS.ACT - a routine which will return the absolute value of an INT.

ALLOCATE.ACT - routines which allow dynamic runtime memory manipulation.

CHARTEST.ACT - routines which perform various tests and functions on characters.

CIRCLE.ACT - a circle drawing routine using neither Sine nor Cosine.

CONSOLE.ACT - a routine which both debounces the console keys and allows you to tie routines into them.

IO.ACT - routines which implement some advanced I/O operations.

JOYSTIX.ACT - routines which make interpreting joystick input easier.

PMG.ACT - player/missile graphics routines.

PRINTF.ACT - an extended version of the ACTION! Library 'PrintF'.

REAL.ACT - routines which allow you to use floating point numbers.

SORT.ACT - QuickSort for BYTE, CARD, INT, and string data.

TURTLE.ACT - an implementation of turtle graphics, ala LOGO.

GEM.DEM - a four person game written in ACTION!.

KALSCOPE.DEM - a colorful demo of ACTION!'s speed.

MUSIC.DEM - a demo which creates a playable organ.

SNAILS.DEM - a two person game translated from BASIC to ACTION!.

WARP.DEM - a one person game which uses many of the advanced constructs and abilities of ACTION!.

There are also some files with the extension '.DMn', where 'n' is a number. These are demos of the routines in a specific file, designed to help you better understand the procedure required to make use of the Tool Kit routines.

Note: In most of the ACTION! source files there are global variables and procedures which contain the underline character ('_'). These variables and routines are internal to the ToolKit routines, and should be neither called nor accessed by you unless you are positive you know for what they are used.

To Boot This Disk simply boot your DOS disk with the ACTION! cartridge inserted, and then put this disk in your drive. THIS DISKETTE DOES NOT HAVE DOS ON IT AND WILL NOT BOOT DIRECTLY.

ABS.ACT

INT FUNC Abs(INT n)

Purpose: To return the absolute value of an INTeger.

Params: n - the INTeger whose absolute value is returned.

Description: This function will return the absolute value of the INT passed to it.

ALLOCATE.ACT

The routines in this file allow you to allocate and free blocks of memory at runtime. If you want to use this capability, you must first call the AllocInit routine. AllocInit expects a global CARD variable called EndProg to contain the address of the end of your program. To do this, compile your program, and then type the following in the Monitor immediately after compiling:

SET EndProg=* [RETURN]

Now you can run your program.

Technical Note: The Alloc and Free routines operate on a 'free list'. This list gives the location and size of every free memory block. Alloc simply removes a block from the free list, and Free puts a block back into the list.

PROC AllocInit(CARD p)

Purpose: To set up the free list and initialize the allocation routines.

Params: p - the address of the first free memory location in memory.

Description: This routine is used to create the free list so that Alloc and Free may be used. See the introduction to this section for instructions on its use.

Note: If you are planning to use P/M graphics and/or bit-map graphics, you should enable the P/Ms and be in the most memory intensive graphics mode you plan to use when you call AllocInit, since it considers all memory up to MEMHI ($2E5) to be free space. (Alternatively, you can merely change the value of MEMHI.)

CARD FUNC Alloc(CARD nBytes)

Purpose: To allocate a block of memory of a specified size, returning the address of that block.

Params: nBytes - the size in bytes of the block to be allocated.

Description: This routine allows you to reserve a block of memory 'nBytes' long. The starting address of the block is returned, so, for example, you could use it to allocate space for a large array at runtime, after you've determined the size array you need:

```
PROC Test()
  CARD size
  BYTE ARRAY bigarray

  Print("Size of Array)) ")
  size=InputC()
  bigarray=Alloc(size)
RETURN
```

Note: the smallest block you can allocate is 3 bytes.

PROC Free(CARD target,nBytes)

Purpose: To free a block of memory which has previously been reserved using the Alloc function.

Params: target - the starting address of the block to free.
nBytes - the length in bytes of the block to free.

Description: This procedure allows you to return a block of memory used by Alloc to the free list.

PROC PrintFreeList()

Purpose: To print out the free list.

Params: none

Description: This procedure will print out the current free list, and should be used mostly for diagnostic debugging reasons.

CHARTEST.ACT

The routines in this file are very diverse, including:

IsAlpha - a character test
IsUpper - a character test
IsLower - a character test
IsDigit - a character test
ToUpper - a character manipulation
ToLower - a character manipulation

BYTE FUNC IsAlpha(BYTE c)

Purpose: To test a single character to see if it is a letter.

Params: c - the character to be tested.

Description: This function checks c to see if it is an alphabetic character. If it is, a 1 is returned; otherwise a 0 is returned.

BYTE FUNC IsUpper(BYTE c)

Purpose: To test a single character to see if it is an uppercase letter.

Params: c - the character to be tested.

Description: This function checks c to see if it is an uppercase alphabetic character. If it is, a 1 is returned; otherwise a 0 is returned.

BYTE FUNC IsLower(BYTE c)

Purpose: To test a single character to see if it is a lowercase letter.

Params: c - the character to be tested.

Description: This function checks c to see if it is a lowercase alphabetic character. If it is, a 1 is returned; otherwise a 0 is returned.

BYTE FUNC IsDigit(BYTE c)

Purpose: To test a single character to see if it is a digit.

Params: c - the character to be tested.

Description: This function checks c to see if it is a digit (0 - 9). If it is, a 1 is returned; otherwise a 0 is returned.

BYTE FUNC ToUpper(BYTE c)

Purpose: To change lowercase letters to uppercase.

Params: c - the character to put into uppercase.

Description: This function will return the uppercase of the character passed to it. If the character is already uppercase, or is not alphabetic, then the character is returned unchanged.

BYTE FUNC ToLower(BYTE c)

Purpose: To change uppercase letters to lowercase.

Params: c - the character to put into lowercase.

Description: This function will return the lowercase of the character passed to it. If the character is already lowercase, or is not alphabetic, then the character is returned unchanged.

CIRCLE.ACT

The circle drawing routine in this file is somewhat special, since it does not need to compute Sine or Cosine, and so is very fast. One caveat, however, this routine does no screen bounds checking, so either make sure your circle will fit on the screen, or add your own bounds checking.

PROC Circle(INT x BYTE y,r,c)

Purpose: To draw a circle of specified center, radius, and color.

Params: x - the horizontal position of the center of the circle to be drawn.
y - the vertical position of the center of the circle to be drawn.
r - the radius of the circle.
c - the color of the circle.

Description: This procedure allows you to draw a circle of specified center, radius, and color. The system variable color is set to c, so c should not be the actual color number as used in the SetColor procedure, but rather the 'color' value in the current graphics mode which corresponds to the SetColor register which contains the color you want to use.

CONSOLE.ACT

The routines in this file allow you to hook the execution of a specific routine to the pressing of one of the console keys (START, SELECT, OPTION). Before you use this capability you must call the InitConsole procedure as follows:

```
InitConsole()
```

Once you have done this you need simply equate the address of your routine to one of the console keys. You can do this in the following manner:

1 - write your routine (it can have no parameters).
2 - call the InitConsole procedure.
3 - equate the name of the console key you wish to your routine.

The following example should help clarify this procedure:

```
INCLUDE "CONSOLE.ACT"

PROC DoStart()
  PrintE("START Pressed")
RETURN

PROC DoSelect()
  PrintE("SELECT Pressed")
RETURN
PROC DoOption()
  PrintE("OPTION Pressed")
RETURN

PROC Main()
  InitConsole()     ;set up console handler
  Start=DoStart    ;DoStart when START
  Select=DoSelect ;DoSelect when SELECT
  Option=DoOption ;DoOption when OPTION
  DO OD
RETURN
```


IO.ACT

The routines in this file allow you to do advanced disk file manipulation from an ACTION! program. Operations implemented are:

Rename a file	Format a diskette
Erase a file	Block Get of data from disk
Protect a file	Block Put of data to disk
Unprotect a file	

Note: The first four of the above operations (those involving a disk file) require that the file name have the 'Dn:' (n=1-8) device specifier prepended to the actual file name; otherwise you will get a 'Nonexistent Device' error.

PROC Rename(BYTE ARRAY filename)

Purpose: To rename a disk file.

Params: filename - the old and new file names.

Description: This routine will rename the specified disk file, and should be used as follows:

```
Rename("D1:TEMP1.ACT TEMP.ACT")
```

This example will rename TEMP1.ACT on drive 1 to TEMP.ACT. Notice that the new name follows the old name in the file name string, with only a space or comma separating the two. Note that the new name may NOT have a device specifier.

PROC Erase(BYTE ARRAY filename)

Purpose: To erase a disk file.

Params: filename - the file to erase.

Description: This procedure will erase a disk file and should be used as follows:

```
Erase("D2:JUNK.ACT")
```

This example will erase JUNK.ACT on drive 2.

PROC Protect(BYTE ARRAY filename)

Purpose: To protect a disk file.

Params: filename - the file to protect.

Description: This will protect a disk file, and should be used as follows:

```
Protect("D1:*.*")
```

This example will protect all files on drive 1.

PROC UnProtect(BYTE ARRAY filename)

Purpose: To unprotect a disk file.

Params: filename - the file to unprotect.

Description: This procedure will unprotect a file which has been protected using either the Protect routine above, or the DOS XL PRO command. It is used in the same way as Protect above.

PROC Format(BYTE ARRAY DriveSpec)

Purpose: To format a diskette.

Params: DriveSpec - the drive containing the disk to be initialized.

Description: This routine allows you to initialize disks, and should be used as follows:

```
Format("D2:")
```

This example will format whatever disk is in drive 2 (unless of course it has a write protect tab on it).

CARD FUNC BGet(BYTE chan CARD addr,len)

Purpose: To read a block of binary or text data from a specified device.

Params: chan - the channel.
addr - the address at which to put the data.
len - the number of bytes of data.

Description: This function allows you to read a block of data, returning the actual number of data bytes read (this will be different from len if End-Of-File was reached before 'len' bytes were read).

PROC BPut(BYTE chan CARD addr,len)

Purpose: To write a block of binary or text data to a specified device.

Params: chan - the channel.
addr - the address from which to get the data.
len - the number of bytes of data.

Description: This procedure allows you to write a block of data, and is the complement to BGet.

JOYSTIX.ACT

INT FUNC HStick(BYTE port)

Purpose: To return the horizontal reading of a specified joystick.

Params: port - the port of the joystick whose horizontal reading is desired.

Description: This routine reads the value of a joystick and returns the following values:

```
-1 - horizontal movement left
 0 - no horizontal movement
 1 - horizontal movement right
```

This routine is much easier to use than the Stick function in the ACTION! Library.

INT FUNC VStick(BYTE port)

Purpose: To return the vertical reading of a specified joystick.

Params: port - the port of the joystick whose vertical reading is desired.

Description: This routine reads the value of a joystick and returns the following values:

```
-1 - vertical movement up
 0 - no vertical movement
 1 - vertical movement down
```

This routine is much easier to use than the Stick function in the ACTION! Library.

PMG.ACT

The routines in this file allow you easy implementation of the ATARI's player/missile (hereafter called P/M's) graphics capabilities. To give you a sense of the extent of this implementation, we'll give a quick synopsis of the routines before going into them in detail:

```
PMGraphics - Set up P/M graphics
PMSetColor - Set a P/M's color
PMAdr      - Give the address of a P/M
PMClear    - Erase a P/M
PMMove     - Move a P/M
PMCreate   - Create a P/M
PMHit      - Test the P/M collision registers
PMHitClr   - Reset the collision registers
PMHPos     - Horzontal positions of P/Ms
PMVPos     - Vertical positions of P/Ms
Graphics   - A modified Graphics
```

Introductory Notes: In several of the routines in this section you will see the parameter num, referring to the number of the player/missile. This number is assigned values as follows:

```
0 - player 0        4 - missile 0
1 - player 1        5 - missile 1
2 - player 2        6 - missile 2
3 - player 3        7 - missile 3
```

In some cases only the values 0 - 3 will be valid or make sense.

PROC PMGraphics(BYTE mode)

Purpose: To turn P/M graphics on or off.

Params: mode - determines which P/M mode.

Description: This procedure is very much like the Graphics routine in the ACTION! Library, except that this one controls player/missile graphics. The mode values are as follows:

```
0 - turn off P/Ms
1 - single line resolution P/Ms
2 - double line resolution P/Ms
```

Note: This procedure moves all the players and missiles off the screen, but does not erase the P/M memory. To erase it, use PMClear.

PROC PMSetColor(BYTE num,hue,lum)

Purpose: To set the hue and luminance of a player and its associated missile.

Params: num - the player number (0-3).
hue - the hue for the player.
lum - the luminance for the player.

Description: This procedure is very much like the SetColor Library routine. In fact the colors corresoonding to hue and lum are exactly as shown in the ACTION! manual under SetColor. The difference is that it allows you to set the color of a P/M, not a playfield.

CARD FUNC PMAdr(BYTE num)

Purpose: To return the address of a given P/M's memory block.

Params: num - the P/M number.

Description: This function returns the starting address of the memory block allotted to the P/M specified by num. Since the missiles all occupy the same block of memory, num values 4 - 7 will all return the same address.

PROC PMClear(BYTE num)

Purpose: To clear out the memory block of a specified P/M.

Params: num - the P/M number.

Description: This procedure zeroes all the bytes in the memory block of the P/M given by num. If it is a missile, only that part of the block allotted to the missile will be zeroed.

PROC PMMove(BYTE num,x,y)

Purpose: To move a specified P/M

Params: num - the P/M number.
x - horizontal position to which to move the P/M.
y - vertical position to which to move the P/M.

Description:This procedure allows you to move P/Ms easily and quickly. You simply need to specify the P/M number and the x,y position to which you want it moved.

PROC PMCreate(BYTE num BYTE ARRAY pm BYTE len,width,x,y)

Purpose: To allow easy creation of a P/M.

Params: num - the P/M number.
pm - the array which contains the P/M's shape data.
len - the length of the array pm.
width - the width of the player.
x - the starting horizontal position of the P/M.
y - the starting vertical position of the P/M.

Description: This routine allows you to create a P/M. You need to pass it the P/M number, the name of the array which contains its shape, the length of that array,the P/M's width (1=single, 2=double, 4=quadruple), and the starting x,y position of the P/M.

BYTE FUNC PMHit(BYTE num,cnum)

Purpose: To determine whether a specified P/M has collided with a specified player or playfield.

Params: num - the P/M number.
cnum - the player or playfield to test for a collision.

Description: This function allows you to see if a given P/M has collided with a specified player or playfield, returning a 1 if there is a collision, a 0 otherwise. The num values are described in the beginning of this section, but the cnum values need to be explained:

0 - player 0	8 - playfield 0
1 - player 1	9 - playfield 1
2 - player 2	10 - playfield 2
3 - player 3	11 - playfield 3

The playfield numbers 0-3 are the same as those used in the SetColor Library routine to set playfield colors.

BYTE PMHitClr

Purpose: To clear the P/M collision registers.

Params: Not Applicable.

Description: By using the statement: PMHitClr=0 you can clear the P/M collision registers. You should do this just before you do something which might result in a collision (such as PMMove), or you may have information from previous collisions still in the registers.

BYTE ARRAY PMHPos(8)

Purpose: To keep track of the current horizontal positions of the P/Ms.

Params: The element number of the array (same as P/M number).

Description: By accessing an element of this array you can find out the current horizontal position of any P/M. Simply use the P/M number as the array element (e.g. PMHPos(3) will give the horizontal position of player 3). The values in this array should not be changed by you.

BYTE ARRAY PMVPos(8)

Purpose: To keep track of the current vertical positions of the P/Ms.

Params: The element number of the array (same as P/M number).

Description: By accessing an element of this array you can find out the current vertical position of any P/M. Simply use the P/M number as the array element (e.g. PMVPos(5) will give the vertical position of missile 1). The values in this array should not be changed by you.

PROC Graphics(BYTE mode)

Purpose: To turn off P/M graphics whenever changing bit-map graphics modes.

Params: mode - same as in the Graphics Library routine.

Description: This procedure simply turns off the P/M graphics every time you change bit-map graphics modes, and replaces the normal Graphics Library routine. This routine is necessary, for the P/M graphics memory is allocated just below screen memory, so changing screen modes could wipe out part of the P/M space. If you are changing between graphics modes which use the same amount of memory, you can comment out this procedure from the source listing and so keep Graphics just the way it was.

PRINTF.ACT

The following two procedures are extensions of the Library PrintF routine, and allow you to control field size and justification as well as the type of data output.

The following routines are internal to the PRINTF routines, and should not be used by you unless you are sure of their function:

```
BYTE FUNC PF_ToLower
BYTE FUNC PF_IsDigit
CARD FUNC PF_Nbase
```

PROC PrintF(BYTE ARRAY control CARD c1,c2,c3,c4,c5,c6)

Purpose: To allow formatted output of data.

Params: control - the string which determines the format of the following data.
c1 thru c6 - the data to be output

Description: This procedure is an upgrade to the PrintF routine in the ACTION! Library. The difference lies in the controls available and the modifcations which can be made to the controls. The controls themselves are:

```
%D - Decimal Notation
%O - Octal Notation
%H - Hexadecimal Notation
%U - Unsigned CARD Notation
%C - Character
%S - String (BYTE ARRAY)
%E - Carriage Return/End-of-Line
%% - the '%' character
```

So far this looks very similar to the 'normal' PrintF routine. However, the best is yet to come. Between the '%' and the control character (except 'E' and '%') you may insert some field size and justification information as follows:

A minus sign: this indicates left justification of the data within its field (right justification is the default).

A number: determines the minimum field size for the data. The data will be printed in a field at least number wide, and wider if the data is too long. If the data is shorter than the field size it will be right justified in the field unless the '-' modifier has been used.

A '.' followed by a number: indicates the maximum number of characters of data to print into the field.

Example: The following list of control strings show how the different modifiers affect the printing of the string "ACTION!" (we have placed broken bars to show the field size):

```
%S        |ACTION!|
%5S       |ACTION!|
%10S      |   ACTION!|
%-10S     |ACTION!   |
%10.4S    |      ACTI|
%-10.4S   |ACTI      |
%.4S      |ACTI|
```

PROC PrintFD(BYTE chan BYTE ARRAY control CARD c1,c2,c3,c4,c5,c6)

Purpose: To allow formatted output of data to a specified channel.

Params: chan - the channel number (0-7)
control - same as PrintF
c1 thru c6 - same as PrintF

Description: This procedure is exactly like the above PrintF, except that it allows you to direct the output to a specific channel (device).

REAL.ACT

This file contains routines which allow you to access the ROM floating point routines from ACTION!, thus making the ACTION! language more useful when writing numerically oriented programs.

To use the floating point routines (hereafter called the Real routines), you must declare variables of the type REAL, for example:

```
REAL x,y,z
```

The type REAL is actually a record type, so the name of the variable is a pointer to the record itself. This makes it very similar to an array.

You cannot use the assignment statement to assign a value to a real, since the ACTION! Compiler does not internally understand reals. You must instead use RealAssign, ValR, IntToReal, InputR, or InputRD.

Also included in this file are some mathematical routines to manipulate reals, as well as routines to print out reals.

Following each routine's description section are some examples of that routine's usage. For these examples, assume the following declarations:

```
REAL xreal,yreal,zreal

BYTE ARRAY astring

INT xint,yint,zint

BYTE channel
```

The following routines are internal to the ACTION! real routines, and should not be used by you:

```
PROC  ROM_AFP       PROC ROM_FASC
PROC  ROM_IFP       PROC ROM_FPI
PROC  ROM_FSUB      PROC ROM_FADD
PROC  ROM_FMULT     PROC ROM_FDIV
PROC  ROM_EXP       PROC ROM_EXP10
PROC  ROM_LOG       PROC ROM_LOG10
PROC  ROM_INIT
```

Note: You will often see the type REAL POINTER in the declaration of the parameters of a routine. This simply means that you should use the name (identifier) of the real, since the name alone is a pointer to the real.

REAL Conversion Routines

PROC IntToReal(INT i REAL POINTER r)

Purpose: To put an INT value into a REAL variable.

Params: i - the INT value to be assigned to the REAL.
r - the REAL to which the INT value is assigned.

Description: This procedure allows you to assign the value of an INT to a REAL variable. If the ACTION! compiler could manipulate reals, this routine would be the equivalent of: r=i.

Examples:

```
xint=453
IntToReal(xint,xreal) ;xreal now equals 453
IntToReal(2534,yreal) ;yreal now equals 2534
```

INT FUNC RealToInt(REAL POINTER r)

Purpose: To return the INT value of a REAL variable.

Params: r - the REAL variable.

Description: This function will return the INT value of the REAL passed to it as a parameter.

Examples:

```
xint=RealToInt(xreal) ;xint now equals the INT value of xreal
```

PROC StrR(REAL POINTER r BYTE ARRAY s)

Purpose: To convert a REAL to a string.

Params: r - the REAL to convert.
s - the string in which to store the character representation of the REAL.

Description: This procedure converts a REAL into its character representation.

Examples:

```
InToReal(3926,xreal) ;xreal = 3926
StrR(xreal,astring)  ;astring now contains "3926"
```


PROC ValR(BYTE ARRAY s REAL POINTER r)

Purpose: To convert a string to a REAL.

Params: s - the string to convert.
r - the REAL to which the value of s will be assigned.

Description: This procedure will convert as much of the string as possible into a REAL variable (i.e., if the string is "abcde", this routine will put 0 into the REAL).

Examples:

```
astring="45.276"
ValR(astring,xreal)  ;same as xreal=45.276
ValR("2.7E-4",yreal) ;same as yreal=2.7*10^-4
ValR("70.2agr",zreal);same as zreal=70.2
```

REAL Mathematical Routines

PROC RealAssign(REAL POINTER a,b)

Purpose: To assign the value of one REAL variable to another.

Params: a - the REAL value to assign.
b - the REAL to which the value a is assigned.

Description: This procedure allows you to assign the value of one REAL to another one. If the ACTION! Compiler could manipulate reals, the equivalent would be: b=a.

Examples:

```
RealAssign(xreal,yreal) ;same as yreal=xreal
RealAssign(zreal,yreal) ;same as zreal=yreal
```

PROC RealAdd(REAL POINTER a,b,c)

Purpose: To add two REALs

Params: a - an addend
b - an addend
c - the sum

Description: This procedure allows you to add two REALs. If the ACTION! Compiler could manipulate reals, this routine would equivalent to: c=a+b.

Examples:

```
RealAdd(xreal,yreal,zreal) ;same as zreal=xreal+yreal
```

PROC RealSub(REAL POINTER a,b,c)

Purpose: To subtract two REALs

Params: a - the subtrahend
b - the minuend
c - the difference

Description: This procedure allows you to subtract two REALs. If the ACTION! Compiler could manipulate reals, this routine would equivalent to: c=a-b.

Examples:
RealSub(xreal,yreal,zreal) ;same as zreal=xreal-yreal

PROC RealMult(REAL POINTER a,b,c)

Purpose: To multiply two REALs

Params: a - the multplicand
b - the multiplier
c - the product

Description: This procedure allows you to multiply two REALs. If the ACTION! Compiler could manipulate reals, this routine would equivalent to: c=a*b.

Examples:
RealMult(xreal,yreal,zreal) ;same as zreal=xreal*yreal

PROC RealDiv(REAL POINTER a,b,c)

Purpose: To divide two REALs

Params: a - the dividend
b - the divisor
c - the quotient

Description: This procedure allows you to divide two REALs. If the ACTION! Compiler could manipulate reals, this routine would equivalent to: c=a/b.

Examples:
RealDiv(xreal,yreal,zreal) ;same as zreal=xreal/yreal

PROC Exp(REAL POINTER a,b)

Purpose: To raise e to the a power.

Params: a - the power to which to raise e.
b - the result of raising e to the a power.

Description: This procedure allows you to get the base e exponential of a REAL. The equivalent of this is: $b=e^a$.

Examples:
Exp(xreal,yreal) ;yreal=e^{xreal}

Expi0(REAL POINTER a,b)

Purpose: To raise 10 to the a power.

Params: a - the power to which to raise 10.
b - the result of raising 10 to the a power.

Description: This procedure allows you to compute the base 10 exponential of a REAL. Its equivalent is: $b=10^a$.

Examples:
Expi0(xreal,yreal) ;yreal=10^{xreal}

PROC Power(REAL POINTER a,b,c)

Purpose: To raise a REAL to a REAL power.

Params: a - the base of the power.
b - the power to which to raise a.
c - the result of raising a to the b power.

Description: This routine allows you to raise one REAL to a power specified by another REAL, and is equivalent to: $c=a^b$.

Examples:
Power(xreal,yreal,zreal) ;zreal=$xreal^{yreal}$

PROC Ln(REAL POINTER a,b)

Purpose: To take the natural logarithm of a REAL.

Params: a - the REAL whose natural log is taken.
b - the result of taking the natural log of a.

Description: This procedure allows you to take the natural (base e) logarithm of a REAL, and is equivalent to: b=ln(a).

Examples:
```
Ln(xreal,yreal) ;yreal=ln(xreal)
```

PROC Log10(REAL POINTER a,b)

Purpose: To take the common (base 10) logarithm of a REAL.

Params: a - the REAL whose common log is taken.
b - the result of taking the common log of a.

Description: This procedure allows you to take the common (base 10) logarithm of a REAL, and is equivalent to: b=log(a).

Examples:
```
Log10(xreal,yreal) ;yreal=log(xreal)
```

I/O Routines

PROC PrintR(REAL POINTER a)

Purpose: To output a REAL to the default device.

Params: a - the REAL to be output.

Description: This procedure outputs a real number to the default device without a RETURN.

PROC PrintRD(BYTE channel REAL POINTER a)

Purpose: To output a REAL to a specified channel (device).

Params: channel - the output channel
a - the REAL to be output.

Description: This procedure outputs a real number to the device specified by channel without a RETURN.

PROC PrintRE(REAL POINTER a)

Purpose: To output a REAL to the default device with a RETURN.

Params: a - the REAL to be output.

Description: This procedure outputs a real number to the default device with a RETURN.

PROC PrintRDE(BYTE channel REAL POINTER a)

Purpose: To output a REAL to a specified channel (device) with a RETURN.

Params: channel - the output channel.
a - the REAL to be output.

Description: This procedure outputs a real number to the specified device with a RETURN.

PROC InputR(REAL POINTER a)

Purpose: To input a REAL from the default device.

Params: a - the REAL variable in which to store the input value.

Description: This procedure inputs a real number from the default device and stores it in the specified REAL variable.

PROC InputRD(BYTE channel REAL POINTER a)

Purpose: To input a REAL from a specified channel (device).

Params: channel - the input channel.
a - the REAL variable in which to store the input value.

Description: This procedure inputs a real number from the specified device and stores it in the given REAL variable.

SORT.ACT

The following four sort routines all use the QuickSort algorithm. This algorithm was used because it is very fast (order N log N). In the best case QuickSort is, in fact, among the fastest sorting algorithms known. For comparison, both the Bubble and the Shell algorithms are of order N^2. The QuickSort can deteriorate to this speed when sorting presorted data.

If you take a look at the SORT.ACT source you will see that you can create your own routines to sort REALs or complex record TYPEs simply by writing your own Compare and Swap routines.

Usage Note: Before using any of these routines you should first change the source line which reads

```
DEFINE SortMax="10000"
```

to the maximum size of the data array you expect to encounter. An alternative is to change the sort routines so that they INCLUDE ALLOC.ACT and dynamically create the 'List' array.

PROC SortB(BYTE ARRAY data CARD len BYTE order)

Purpose: To sort one-byte data in either ascending or descending order.

Params: data - the array containing the data to be sorted.
len - the length of the data array.
order - determines order of sort (0=ascending, 1=descending)

Description: This procedure allows you to sort one-byte data very quickly.

PROC SortC(CARD ARRAY data CARD len BYTE order)

Purpose: To sort two-byte unsigned data in either ascending or descending order.

Params: data - the array containing the data to be sorted.
len - the length of the data array.
order - determines order of sort (0=ascending, 1=descending)

Description: This procedure allows you to sort two-byte unsigned data very quickly.

PROC SortI(INT ARRAY data CARD len BYTE order)

Purpose: To sort two-byte signed data in either ascending or descending order.

Params: data - the array containing the data to be sorted.
len - the length of the data array.
order - determines order of sort (0=ascending, 1=descending)

Description: This procedure allows you to sort two-byte signed data very quickly.

PROC SortS(CARD ARRAY data CARD len BYTE order)

Purpose: To sort string data in either ascending or descending order.

Params: data - the array containing the addresses of the strings to be sorted.
len - the length of the data array.
order - determines order of sort (0=ascending, 1=descending)

Description: This procedure allows you to sort strings very quickly. Notice that the addresses of the strings to be sorted must be the elements of the CARD ARRAY data.

TURTLE.ACT

The routines in this file implement turtle graphics ala LOGO.

These routines require that the screen be in a bit-map graphics mode in which Plot and DrawTo are useable. Also, the length of a line drawn depends on the graphics mode, and there is no screen bounds checking.

Also, the color of the line drawn by the turtle depends entirely upon then current value of the system variable color, so you should use SetColor and color to choose the color you want.

The following routines are internal to the turtle graphics and should not be called by you

```
CARD FUNC TG_ISin     CARD FUNC TG_ICos
```

PROC Right(INT theta)

Purpose: To turn the turtle right (clockwise) theta degrees.

Params: theta - the angle to turn the turtle clockwise.

Description: This procedure allows you to turn the turtle clockwise a specified number of degrees.

PROC Left(INT theta)

Purpose: To turn the turtle left (counterclockwise) theta degrees.

Params: theta - the angle to turn the turtle counterclockwise.

Description: This procedure allows you to turn the turtle counter-clockwise a specified number of degrees.

PROC Turn(INT theta)

Purpose: To turn the turtle either clockwise or counterclockwise.

Params: theta - the angle to turn the turtle.

Description: This routine allows you to turn the turtle either clockwise or counterclockwise, depending on the sign of the angle. If theta is positive, the turtle will turn counterclockwise, otherwise it will turn clockwise.

PROC Forward(INT length)

Purpose: To move the turtle forward a specified length.

Params: length - the length to move forward.

Description: This procedure allows you to move the turtle forward a specified length. This length depends entirely upon the current graphics mode.

PROC SetTurtle(INT x,y,theta)

Purpose: To move the turtle to a specified x,y position at a given angle.

Params: x - the horizontal position at which to set the turtle.
y - the vertical position at which to set the turtle.
theta - the angle at which to set the turtle.

Description: This procedure allows you to move the turtle to an absolute x,y position and point it in a specific direction. At theta=0° the turtle points right, at 90° it points up, at 180° it points left, and at 270° it points down. In essence, increasing positive values of theta turn the turtle counterclockwise, and increasing negative theta values turn it clockwise.

GEM.DEM

Gem is a game which was written by Joel Gluck after having the ACTION! cartridge for only 2 days. If you look at the code, you will notice how similar it is to BASIC. This reflects Joel's previous programming experience (BASIC only) and is not due to its being originally written in BASIC (which it was not). Enough of its history. Gem is designed for 1 to 4 players, each using a joystick. The object is to steal the gem in the center of the screen and return to your home base before one of the robots or other players zaps you. However, before the game itself begins, you are prompted for some information, specifically:

How many points to win the game?

How many robots in the final round?

Winning Points - to win a point, you must get the Gem and return with it to your corner.

Robots - the number of robots increase each round. Although they seem to die off, whenever the gem is picked up they all come back. If one of the robots is destroyed while one of the players is carrying the gem, it is reincarnated immediately.

Zapping - to zap a robot or another player, press the joystick trigger while pointing the joystick in the direction of the target. While you are zapping you cannot move. You can also zap by running into the target, but this also zaps you, so only use this method when on a Kamikaze run to keep another player from getting the gem home.

Getting Zapped - when you get zapped, you are reincarnated back at your home base and the gem is taken from you if you are carrying it. There is no limit to the number of times you can be reincarnated.

Winning - when one of the players has accumulated the required number of points, he wins the game, and you may either play again, or quit and go to the ACTION! monitor.

Technical Notes - to use this game, do not read it into the ACTION! Editor and then compile it, since there is not enough memory to do both. Instead, RUN it from the ACTION! Monitor directly from disk. (This note does not apply if you are using DOS XL, since you have more memory and can have the game in the Editor while compiling it).

The maximum recommended number of robots is 100. Bugs will appear in the program if you use many more, but do not fret. The most robots survived to date is 45 in a one player game.

KALSCOPE.DEM

This demo program uses advanced math and display list algorithms to achieve the effect of a kaleidoscope on your TV. When you run it you will be amazed by its speed. You can even change the kaleidoscope's speed and persistence (amount of time a point remains on the screen) by moving joystick 0 vertically or horizontally, respectively. After playing with it a while you will be surprised by the number and variety of the different patterns it can create.

P.S. - you can freeze the picture by pressing the trigger.

MUSIC.DEM

This demo program uses a couple of the Toolkit utilities and knowledge of the Atari's keyboard matrix to produce an organ which will play as you press the keys.

The letters on the keyboard represent the notes, and the letters above and below the keyboard represent the actual computer keys you must press to get the note. By pressing <SHIFT><note> you can access the middle octave, and by pressing <CONTROL><note> you can access the high octave.

This organ is special (for Atari's) in that it only plays a note as long as you keep the key depressed. Few people know how to determine how long a key is pressed (unless they've deciphered the Type-a-Tune demo in the BASIC reference manual, or waded through the hardware manual), so if you look at the source code you can discover something useful (possibly).

SNAILS.DEM

Games similar to "SNAILS' TRAILS" have been around for a long time. A version called "SURROUND" was one of the first games available for the Atari 2600. But, in the tradition of the video game industry, we present a storyline:

You are a giant, mutant snail. Wherever you travel, you leave a trail of radioactive slime behind. So poisonous and impenetrable is this slime that should any being (including you yourself) touch it, it dies instantly. (Yes, yes. If it's that poisonous, how could you lay the trail in the first place? How should we know...YOU are the mutant.)

Further, the scientists of far off H'tra-E have discovered your kind and have imprisioned you and another of your race in a large rectangular arena. Unfortunately, both of you are neither male or female. Instead, you are each a S'ti, specially bred to do battle until death! You don't know the meaning of the word "STOP".

So, as the scientists release you from stasis (you hear three bells as the stasis field is lifted), you begin by charging straight toward your opponent. But wait! A bit of intellingence enters your crazed brain. If your slime trail is so deadly, perhaps you can entice your enemy to run into it, thus killing the other S'ti without damage to yourself. Great strategy!

What's this, though? Your opponent has developed the same strategy. Now you and the other S'ti must race around the arena, with the strategic goal of forcing each other to touch a poisonous trail or to run into the electrified outer fence. (Well, we had to keep you in the arena somehow, didn't we?). But tactics can be important as well. Look, you are running straight across the arena. At the last second, you veer in front of your enemy! He can't avoid your trail in time! He's going to...Oops. You forgot about the wall. Too bad. R.I.P.

To make a long story into a short game, you and another human opponent must each use a joystick (plugged into ports 1 and 2) to control your snail. The first snail to run into a slime trail or a wall loses, and the other snail scores a point. The first snail to score 10 points wins the game. Also, if both snails die at the same instant, neither scores a point. Good Luck!

P.S. This game was converted from BASIC XL to ACTION! in about two hours. The original BASIC XL version is in Chapter 29 of Thirty Days to Understanding BASIC XL and is on the BASIC XL ToolKit diskette.

WARP.DEM

Warp Attack is a game for only the most daring interstellar pilots. You have been chosen as one of this special breed and are sent on a surface patrol over the planet Stripes. You can move you ship left or right and you can dive or climb as in an airplane, but your on-board navigational equipment won't allow you to crash into the planet surface. As you are flying along minding your own business, an Hospites (your sworn enemy) Stellar Fortress warps right into your path, and she's armed to the teeth with SeeKer PlasmaBalls. One touch of them and you're dust. And all you have are puny pulse cannons.

Now you know why only the best were chosen for this assignment: very few know how to destroy a Stellar Fortress, and you are one of them. You first must destroy its right engine (on your left as it approaches), then its left engine, and finally its main engine, and each must be a direct hit. While completing this feat of precision marksmanship you must remember to avoid those PlasmaBalls. Piece of Cake!

Technical Notes: Warp Attack uses quite a few advanced programming techniques, including a modified display list, display list interrupts, vertical blank interrupts, and a block fastdraw. The DLI and VBI together create the scrolling planet surface, and the fastdraw is used to move the Stellar Fortress (it's not a player!).